Reducing Fault Sensitivity of Microprocessor-Based Systems by Modifying Workload Structure

D. Audet[1], S. Masson[1] and Y. Savaria[2]

[1]Université du Québec à Chicoutimi
555 boul. de l'Université, Chicoutimi (Québec), CANADA, G7H 2B1

[2]Ecole Polytechnique de Montréal
P.O. Box 6079, Succ. Centre-Ville, Montréal (Québec), CANADA, H2X 3A7

Abstract

The use of off-the-shelf components in microprocessor-based systems can limit the applicability of a number of hardware fault-tolerance methods. Software techniques offer attractive solutions to improve the reliability of systems operating in a hostile environment. The fault sensitivity of a system running a critical application obviously depends on the application execution time and the amount of memory it uses. This study shows that the program structure also has a significant influence on fault sensitivity. Program characteristics, such as the size and duration of iterative and sequential sections, are required to determine the sensitivity profile. It is shown that, provided data dependency is not affected, one can rearrange the program structure to significantly reduce the average sensitivity of a program. Straightforward analysis of the sensitivity profile allows to estimate the reduction. A simple example of code rearrangement is described and it is shown that a 50% reduction could be achieved with respect to the initial structure. The magnitude of the reduction varies from one application to another.

1: Introduction

A number of critical applications require some form of fault-tolerance in order to provide better reliability, availability and dependability. The way fault-tolerance is included actually depends on the targeted application. Numerous hardware fault-tolerance techniques were developed by many researchers to improve the robustness of digital systems, ranging from memories to dedicated control systems. Recent studies [1,2] addressed the problem from another perspective. Indeed, in many space applications, present policies of space agencies advise the use of commercial circuits (COTS: Circuits Off The Shelf) [1]. This prevents the use of techniques to harden the manufacturing processes needed to produce these circuits and therefore limits the applicability of a number of hardware techniques. In such circumstances, software-based methods can provide more practical solutions. In this paper, we attempt to improve the robustness of microprocessor-based systems by modifying the way they perform their tasks. To achieve this goal, we study how the tasks (or workload) they perform are affected by faults, and then define some rules to increase reliability. Because memory circuits constitute a significant part of microprocessor-based systems and transient faults are more

frequent than permanent faults [3,4,5], we focus on the effects of transient faults in memory when a microprocessor executes some task.

Studying fault sensitivity of programs has received little attention. However, as we will demonstrate, it can be used to improve at little or no additional cost the robustness of a system. Intuition suggests that the sensitivity of a program is directly proportional to its length (both in execution time and in memory space it occupies or uses). However, this is not the only factor influencing the sensitivity of a program. As we will show, the way loops and sequential sections are combined affects fault sensitivity. The influence of workload on fault behavior has been studied previously [6,7,8]. In the first paper, fault prediction models were developed using low-level description of the program (for instance, the instruction set). In the others, the effects of workload on fault and error behaviors were studied. By contrast, we are interested in the effects of faults on the workload, and not the other way around. Indeed, we seek ways to modify the structure of the workload to improve reliability using parameters such as the number and size of loops it contains, the length of sequential sections and other characteristics.

2: Workload sensitivity

Determination of workload sensitivity was performed using software fault injection. It must be mentioned that software fault injection is sometimes used to characterize software reliability by injecting high-level faults (E.g. modifying the content of a variable). Here, we used low-level fault injection (E.g.. changing a bit at some memory location) which models more adequately a system operating in a hostile environment. In this paper, we focus on studying the impact of a single bit-flip. On a selected number of programs, a single fault per execution was injected using an interrupt routine activated by an internal timer of the processor. A similar technique was used in other studies [9] but their goals were to investigate some fault characteristics (such as latency) or the efficiency of fault detection methods. The location and the moment of the injection were varied in order to obtain the sensitivity profiles as a function of time and location of the fault. The outcome of the injection could be one of the three following possibilities: 1) *No error observed;* 2) *Bad results obtained;* 3) *System crash*. An error was "observed" if situation 2 or 3 occurred. To carry out this study, we used a dedicated test bench developed earlier [10,11] for the department of national defense of Canada in order to evaluate some fault-tolerance mechanisms in a multiprocessor environment. In this system, the process of injecting faults in the processors memory and collecting data is fully automated.

The applications studied were divided in two categories: A) *Synthetic programs;* B*) Real programs*. Synthetic programs are in fact programs written only to evaluate the behavior of flow control structures such as loops. Figure 1 shows two examples of synthetic program. A bit-flip in the one of the machine instructions implementing these programs can modify the final value of the variable A, or may cause the processor to execute an incorrect or invalid instruction. In program #1, if the affected instruction has already been executed, the fault will never be observed. For loops such as in program #2, this will depend on which iteration the fault occurs. Indeed, it may not be observed if the fault occurs during the last iteration. For all the previous ones, the fault should be observed.

Apart from these synthetic programs, ten "real" programs were also considered, performing various tasks ranging from the determination of the roots of a mathematical expression, to the determination of the maximum number of queens that can be placed on a chessboard so that they can be safe from capture.

Program #1	Program #2
A = A + 1;	For(k=0; k<250; k++){
A = A + 1;	A = A + 1;
A = A + 1;	: (8 additional times)
: (2495 additional times)	A = A + 1;
A = A + 1;	}
A = A + 1;	

Figure 1

3: Results

Figures 2 and 3 show respectively the sensitivity profile of synthetic program #1 and #2 described in the previous section. Note that a large portion of the 23040 faults injected affected program sections not directly related to the task (operating system functions, …etc). The errors caused by these faults were removed from the histograms by subtracting the results obtained with a single-instruction (dummy) program from those obtained with the synthetic one. Although the duration of programs #1 and #2 is almost identical, their sensitivity profiles are radically different. The negative slope of Fig. 2 can be understood by noting that, as the program progresses, the number of instructions and the amount of data that can be affected by the injected fault decreases. Therefore, the program is more sensitive at the beginning and less sensitive at the end. This can be modeled relatively easily. Indeed, let *N* represent the number of instructions of the program, and *P(i,k)* the probability of observing an error when a fault is injected in instruction *i* while the processor is executing instruction *k*.

$$P(i,k) = \begin{cases} 1 & \text{if } i \geq k \quad \text{i.e. injection is in the active set} \\ 0 & \text{if } i < k \quad \text{i.e. injection is in the inactive set} \end{cases}$$

We define the "active set" as the set of instructions the processor must execute before completing its execution and the "inactive set", the set of instructions already executed or that will not be executed. Since *k* is a measure of time, plotting the summation over *i* of *P(i,k)* will produce a profile similar to the one shown in Fig. 2 for a sequential program. For program #2 (Fig. 3), the sensitivity remains relatively constant throughout the execution, since the instructions of the loop are constantly reused, except for the last iteration of the loop. From these two figures, one can assert interesting conclusions:

a) In order to decrease the mean sensitivity of a program, iterative computations (loops) should be placed toward the end of a program whenever possible (that is if data dependency allows this displacement). The reduction in error rate depends on the relative duration of sequential code and iterative code.

b) If a program consists of a large endless loop, a simple way to decrease significantly the sensitivity of the program would be to reload the instruction area of a program at the end of each iteration (leaving the data area untouched). This, in fact, converts the loop into sequential code which reduces the mean sensitivity by approximately 50%. Of course, this implies that there exists a way to obtain a fault-free copy of the program and that one must suffer some performance degradation while executing the task.

c) One can predict the sensitivity profile of a program by looking at its flow control structure (location and size of loops, location and size of sequential portions of the program).

Statements (a) and (b) directly follows from Figures 2 and 3. Statement (c) applies to programs composed of a sequence of loops and sequential computations. As an example, Fig. 4 shows the sensitivity profile of a synthetic program containing two identical loops. As one can see, the curve is made of two steps corresponding to the two loops of the program. This suggests a fourth conclusion that can also help reducing the workload sensitivity:

d) If data dependency is not affected, breaking a large loop into a series of smaller loops can reduced significantly the workload sensitivity.

Complex profiles can be deduced from the flow control structure. Fig. 5 presents the sensitivity profile of a program containing a sequential part and a loop. It is of interest to note that experimental results showed that the nature of the computations performed within a loop (i.e. sequential code or other loops nested inside) does not affect the shape of the sensitivity profile. For instance, if a loop is composed of several loops, the sensitivity profile of a program will, in general, be dominated by the outer loop (as if the inner loops were converted into sequential code). Fig. 6 shows the sensitivity profile of one of the ten real programs selected, that is a FFT. From this profile, one can deduced its structure. Indeed, it is composed of a long loop followed by three smaller ones, each separated by sequential code sections. The last section is relatively longer the previous ones.

By observing Fig. 5, one can predict what would be the sensitivity profile if the *for* loop was placed before the sequential section (consisting of 1250 times the "a=a+1" instruction). Let us call this other program the "program B". For program B, the plateau located at the bottom right of Fig. 5 would be located at the top left, and the number of errors observed would be equal to 240, that is the number of errors observed at $t=0$ in Fig. 5. This means that the number of errors observed would remain equal to 240 from $t=0$ to $t=4915$. After that, the number of errors would decrease linearly until it reaches 0 at $t=11470$. The sensitivity profile of program B would be therefore radically different from that of Fig. 5. A straightforward computation of the area under the curve divided by the total execution time gives the average number of errors observed. For the code used in Fig. 5, the average number of errors is approximately equal to 80 which is about half of what would be observed for program B. In the latter case, the average number of errors would be equal to 170. We see that there is a significant gain to achieve by rearranging the code of program B if data dependency allows. If data dependency prevents rearrangement, it may still be possible to reduce sensitivity. Indeed, it is interesting to note that, if program B is repeated indefinitely, it is possible to make the sensitivity profile identical to that of Fig. 5 by reloading the instruction area of the program at $t=4915$. This strategy would be very effective on programs such as the one presented in Fig. 6.

An attempt was made to determine the lowest fault injection rate required to obtain a reasonable estimate of program sensitivity. For the 10 real programs and the 7 synthetic programs, a fault injection rate of approximately 4×10^{-2} fault per byte of memory could be considered as a lower bound. Since the number of errors observed should be directly proportional to the injection rate, this lower bound was obtained by determining when this relationship ceased to exist. Interestingly, a study of programs executing the same task but having different sizes showed that the sensitivity is not directly proportional to the program size and duration. Fig. 7 shows the sensitivity of loops of different lengths but carrying out the same amount of computations. The circles represent actual measurements. For instance, the circle located at the top right of the curve represents was observed for a loop performing one iteration ($z=1$) consisting in a sequence of 2500 instructions "a=a+1". The next circle on its left represents what was observed for a loop executing two iterations ($z=2$) consisting in a sequence of 1250 instructions "a=a+1". If the sensitivity had been directly proportional to the size of the code, a straight line should have been observed. However, that is not the case. A

model was developed in order to estimate the mean sensitivity *S* of a program. Let us define the sensitivity S(t) as

$$N_{observed}(t) = S(t) N_{injected}(t)$$

where

$N_{injected}$ (t): total number of faults injected at time *t* (one fault injected per execution)
$N_{observed}$ (t): total number of errors observed when faults are injected at time *t*.

If the probability of occurrence of a fault is constant over time for every memory location, one can show that the mean value of *S(t)* can be expressed as a function of the duration of the *I* iterative sections and *J* sequential sections present in the program, that is

$$\overline{S} = S_{t=0} \left[\sum_{i=1}^{I} \left[X_i \sum_{k=1}^{i} G_k \right] + \frac{1}{2} \sum_{j=1}^{J} Y_j \right]$$

where

$\overline{S}$: mean program sensitivity. It represents the probability of observing an error when a fault occurs.
I : total number of iterative sections in the program
J : total number of sequential sections in the program
X_i : ratio of the execution time of the iterative section *i* to the total execution time
Y_j.: ratio of the execution time of the sequential section *j* to the total execution time
G_m : ratio of the size of the sequential section *m* to the total program size.

The last equation is plotted in Fig. 7 as a solid line. Note that, in this figure, the sensitivity of single loops is given. We must mention that, in a loop, there is one iterative section (X_1 is associated to the first *z-1* iterations) and one sequential section (Y_1 is associated to the last iteration). The figure clearly shows what can be understood intuitively, that is if a loop can be used to replace a sequential section of a program, it should be done to reduce the probability of errors due to transient faults. Moreover, the smaller the loop is, the lower will be its sensitivity. However, the sensitivity reduction will not be directly proportional to the size reduction.

4: Conclusion

A study of the effect of transient faults on the workload of microprocessor-based systems has been carried out. It allowed to define a methodology for deducing the sensitivity profile of a workload as a function of its flow control structure. A number of rules aimed at reducing the fault sensitivity of a workload were extracted from experiments on synthetic and real programs. It was shown that, for a simple program structure, a sensitivity reduction of more than 50% can be achieved. The magnitude of the reduction obviously depends on the actual program structure and therefore may vary widely from one application to another. These rules, whenever applicable, can be combined at little or no cost to any hardware fault-tolerance mechanism.

Acknowledgments

This work was supported in part by the department of National Defense of Canada under Contract W2207-4-AF06 and by the Natural Sciences and Engineering Research Council (NSERC) of Canada by Grant OGP0089829.

References

[1] P. Cheynet et al. “Flight Results Analysis of Digital Experiment Devoted to Satellite Image Processing by Means of Neural Nets,” Proc. IEEE Int. On-Line Tesing Workshop, Capri, Italy, July 6-8, 1998, pp. 23-27.

[2] R. Velazco et al. “Artificial Neural Network Robustness for On-Board Satellite Image Processing: Results of SEU Simulations and Ground Tests,” IEEE Trans. on Nuclear Science, Part I, Vol. 44, 1997, pp. 2337-2345.

[3] D.P. Siewiorek and R.S. Schwartz, “Reliable Computer Systems,” Digital Press, Bedford MA, 1992.

[4] R. Horst et al., “The Risk of Data Corruption in Microprocessor-Based Systems,” Proc. IEEE Fault Tolerant Computing Symp. 23, 1993, pp. 576-585.

[5] J.G. Tront et al. “Software Techniques for Detecting Single-Event Upsets in Satellite Computers,” IEEE Trans. on Nuclear Science, Vol. 32, no. 6, Dec. 1985, pp. 4225-4228.

[6] E.W. Czeck, and D.P. Siewiorek, “Observations on the Effects of Fault Manifestation as a Function of Workload,” IEEE Trans. Comput., vol. 41, no. 5 May 1992, pp. 559-566.

[7] P. Duba and R. Iyer, “Transient Fault Behavior in a Microprocessor: A Case Study,” Proc. Int. Conf. Comput. Design (ICCD), Computer Systems Group, Univ. of Illinois, Oct. 1988, pp.272-276.

[8] G. Choi et al. “A Fault Behavior Model for an Avionic Microprocessor: A Case Study,” Proc. Int. Working Conf. Dependable Comput. For Critical Appl., Aug. 1989, pp. 71-77.

[9] G.A. Kanawati et al. “FERRARI: A Flexible Software-Based Fault and Error Injection System,” IEEE Trans. Comput., Vol. 44, no. 2, Feb. 1995, pp. 248-260.

[10] D. Audet, N. Gagnon, and Y. Savaria, "Implementing Fault-Injection and Tolerance Mechanisms in Multiprocessor Systems," IEEE Int. Symp. Defect and Fault Tolerance, Boston (USA), Nov. 6-8, 1996, pp.310-317.

[11] D. Audet, N. Gagnon, and Y. Savaria, "Quantitative Comparisons of TMR Implementations in a Multiprocessor System," IEEE Int. On-Line Testing Workshop, Biarritz (France), July 1996, pp. 196-199.

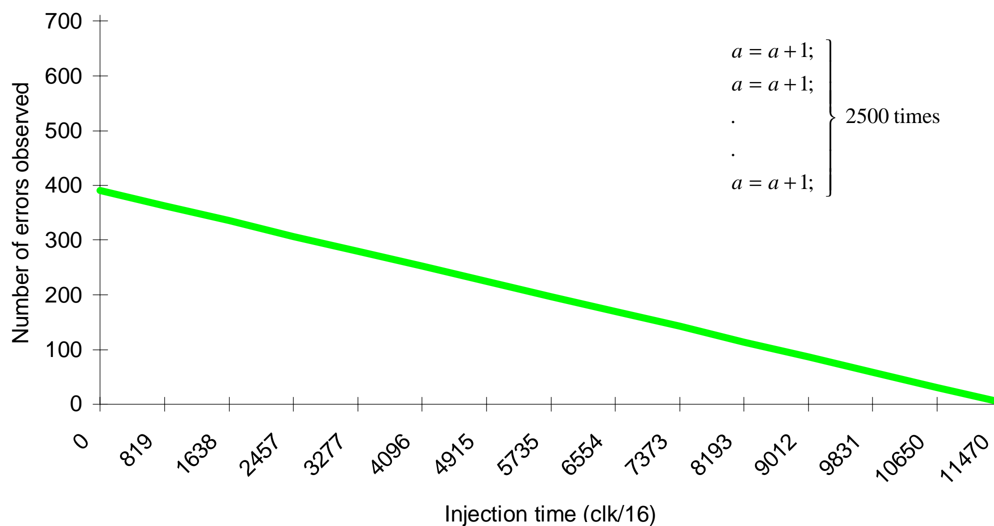

Figure 2

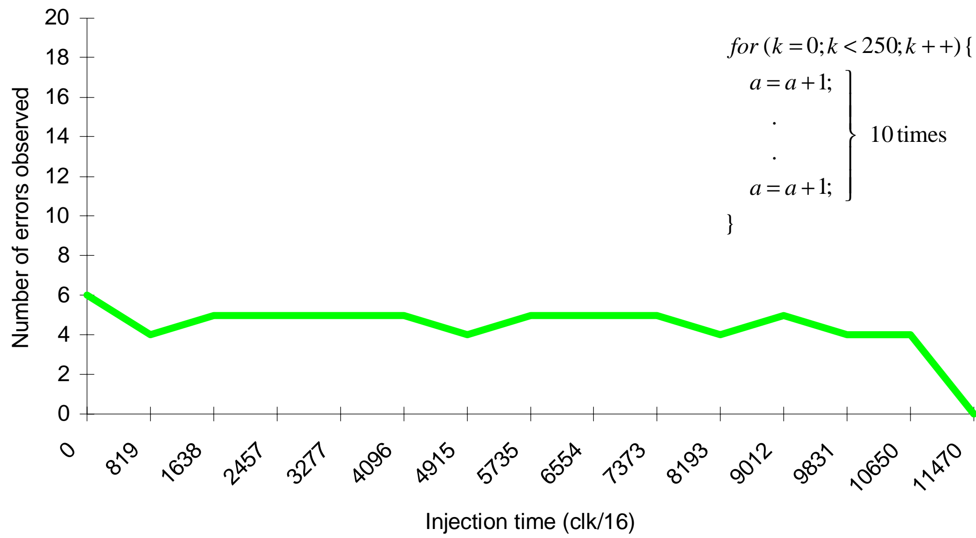

Figure 3

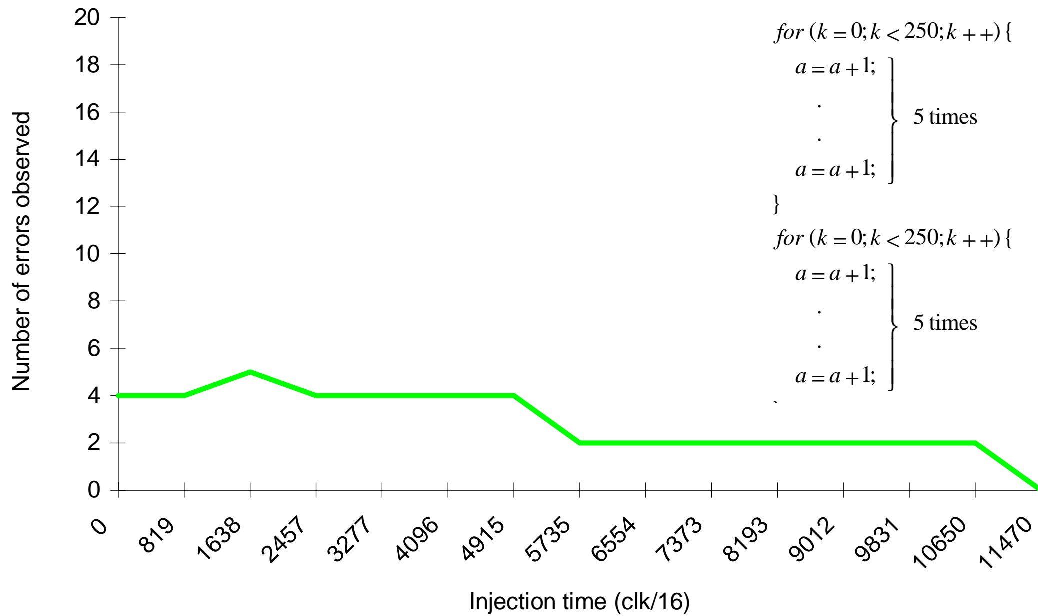

Figure 4

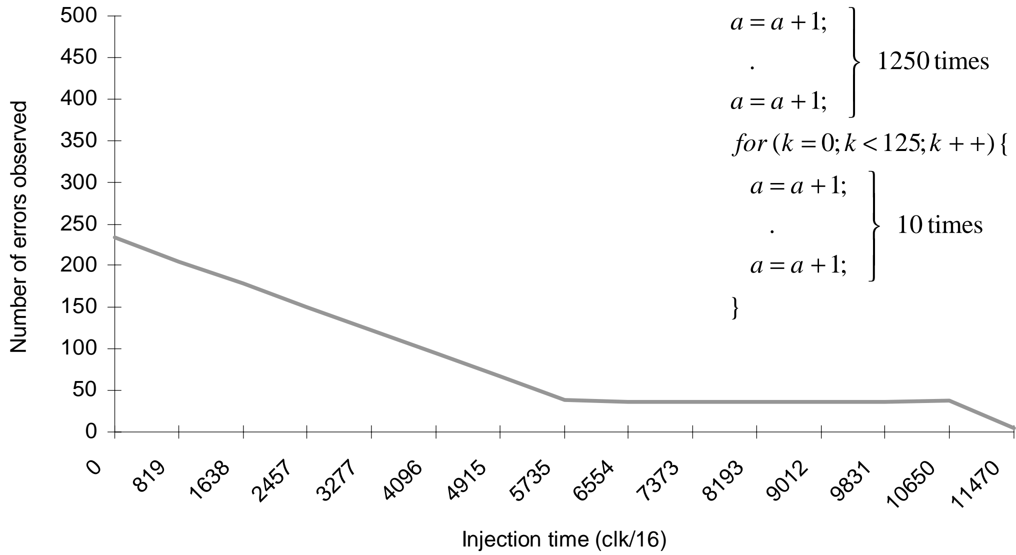

Figure 5

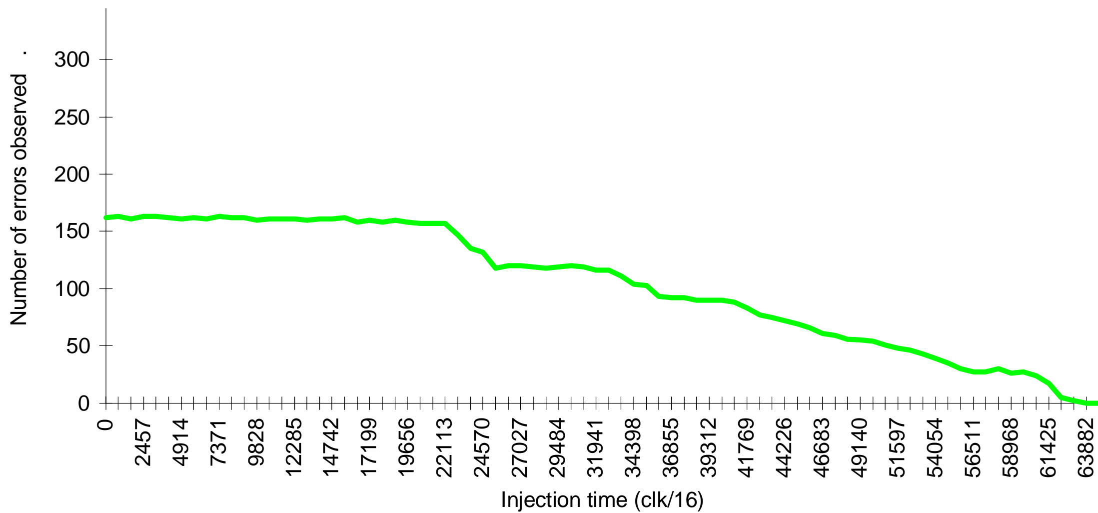

Figure 6

Histogram of errors observed as a function
of the percentage of sequential code
(23 040 executions, 1 fault per execution)

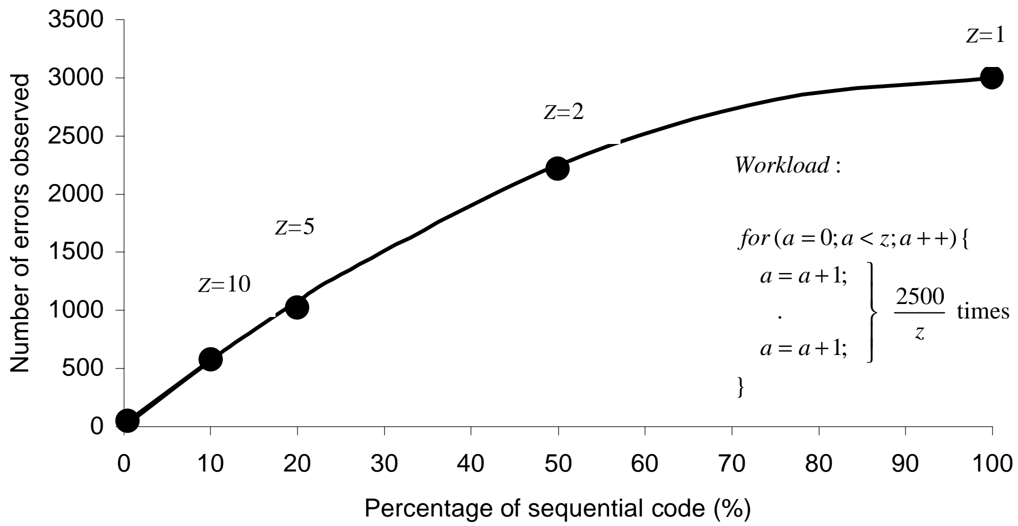

Figure 7